# UNIVERSITÉ PARIS SACLAY
# Master E3A – SETI

# T1 Architecture avancée
# TD1 Processeurs scalaires, superscalaires et VLIW

## 1 Prédiction de branchement

Un branchement a le comportement suivant, où `N` signifie non pris, `P` signifie pris et `P*`$k$ indique une suite de $k$ branchements pris.
`PPNP*`$k$`NP*`$k$`NP*`$k$`NP*`$k$`NP*`$k$...

**1.1** Après la phase d'initialisation, quelle est le pourcentage de bonnes prédictions ?
a) avec un prédicteur 1 bit
b) avec un prédicteur 2 bits.
On considérera les cas où $k = 1$, $k = 3$, $k = 5$.

**1.2** Même question avec un prédicteur un bit et un bit d'historique (initialisé à `N`).

## 2 Processeurs scalaires et superscalaires : exécution de boucles

Soit un processeur superscalaire à ordonnancement statique qui a les caractéristiques suivantes :

- Les instructions sont de longueur fixe (32 bits).
- Il a 32 registres entiers (dont `r0`=0) de 32 bits et 32 registres flottants (de `f0` à `f31`) de 32 bits.
- Il peut lire et exécuter 4 instructions par cycle.
- L'unité entière contient deux pipelines d'exécution entière sur 32 bits, soit deux additionneurs, deux décaleurs. Tous les court-circuits possibles sont implantés.
- L'unité flottante contient un pipeline flottant pour l'addition et un pipeline flottant pour la multiplication.
- L'unité *load/store* peut exécuter jusqu'à deux chargements par cycle, mais ne peut effectuer qu'un *load* et un *store* simultanément. Elle ne peut effectuer qu'un seul *store* par cycle.
- Il dispose d'un mécanisme de prédiction de branchement qui permet de « brancher » en 1 cycle si la prédiction est correcte.

La table 1 donne

- les instructions disponibles
- le pipeline qu'elles utilisent : E0 et E1 sont les deux pipelines entiers, FA est le pipeline flottant pour l'addition et FM le pipeline flottant pour la multiplication.
  Les instructions peuvent être exécutées simultanément si elles utilisent chacune un pipeline séparé.

L'ordonnancement est statique. Les chargements ne peuvent pas passer devant les rangements en attente.
La colonne latence donne la latence entre une instruction source et une instruction destination, dans le cas de dépendances de données. La valeur 1 est le cas où les deux instructions peuvent se succéder normalement, d'un cycle $i$ au cycle $i + 1$.
Soit le programme C suivant (SAXPY) où `x` et `y` sont des vecteurs et `a` est un scalaire de nombres flottants simple précision.

| JEU D'INSTRUCTIONS (extrait) | | | | |
|---|---|---|---|---|
| Mnémo | Exemple | Latence | Pipeline | Effet |
| lf | lf fi, dép.(ra) | 2 | E0 ou E1 | fi ← mem[ra + dépl. (16 bits avec ext. sig.)] |
| sf | sf fi, dép.(ra) | 1 | E0 | fi → mem[ra + dépl. (16 bits avec ext. sig.)] |
| add | add rd,ra, rb | 1 | E0 ou E1 | rd ← ra + rb |
| addi | addi rd, ra, imm | 1 | E0 ou E1 | rd ← ra + imm (16 bits avec ext. sig.) |
| sub | sub rd, ra, rb | 1 | E0 ou E1 | rd ← ra - rb |
| fadd | fadd fd, fa, fb | 4 | FA | fd ← fa + fb |
| fmul | fmul fd, fa, fb | 4 | FM | fd ← fa × fb |
| beq | beq ri, dépl | 1 | E1 | si ri=0 alors cp ← cp+4 + dépl. |
| bne | bne ri, dépl | 1 | E1 | si ri≠0 alors cp ← cp+4 + dépl. |

TABLE 1 – Instructions disponibles

```
float x[1000], y[1000], a;
main ()
{
  int i ;
  for (i=0; i<1000 ;i++)
    y[i] = y[i] + a*x[i] ;
}
```

On utilisera les registres suivants : r1 pointe sur x[i], r2 pointe sur y[i]. r3 a été initialisé à 1000. f0 contient a. f1 et f2 recevront respectivement x[i] et y[i].

**2.1** On considére dans un premier temps une version scalaire du processeur et des caches parfaits (aucun cycle d'attente mémoire).
Traduire le corps du programme en assembleur.
Faire apparaître les cycles d'attentes dues aux dépendances.
Optimiser par réordonnancement ce code pour limiter les cycles d'attente. Donner l'exécution cycle par cycle de la boucle optimisée et le nombre de cycles par itération.

**2.2** Donner la version déroulée (4 itérations par corps de boucle) avec la version scalaire du processeur en supposant des caches parfaits et déterminer le nombre de cycles par itération de la boucle.

**2.3** Donner l'exécution cycle par cycle de la boucle optimisée non déroulée pour la version superscalaire en considérant des caches parfaits, ainsi que le nombre de cycles par itération de la boucle.

**2.4** Donner l'exécution cycle par cycle de version déroulée (4 itérations par corps de boucle) dans la version superscalaire du processeur en supposant des caches parfaits, et le nombre de cycles par itération de la boucle.

## 3 Processeurs VLIW (optionnel)

L'annexe donne le schéma fonctionnel du processeur VLIW TMS320C62 de Texas Instruments, avec les instructions réparties par unité fonctionnelle et les latences des instructions. Dans cet exercice, on utilisera le TMS320C67, qui a une architecture similaire, les mêmes instructions entières et possède en plus des instructions flottantes dont la répartition dans les unités fonctionnelles et les latences sont également fournies en annexe.

**3.1** Examiner l'exécution du code ci-dessous, dans lequel on suppose que sum est un registre initialisé à 0. Que fait cette fonction ? En étudiant les latences, proposer une méthode pour l'optimiser.

```
LOOP:          ADDSP   x,sum,sum
  ||           LDW     *xptr++,x
  || [cond]    B       cond
  || [cond]    SUB     cond,1,cond
```

**3.2** En utilisant le pipeline logiciel, écrire le code assembleur pour la fonction `dotp` « flottante ». Quel est le temps d'exécution de la fonction ?

```
float dotp(float a[], float b[])
{
  int i;
  float sum;
  sum=0.0f;
  for(i=0; i<100; i++)
    sum += a[i] * b[i];
  return sum;
}
```

**3.3** En utilisant le pipeline logiciel, écrire le code assembleur pour la fonction IIR. Quel est le temps d'exécution de la fonction ?

```
void iir(short x[], short y[], short c1, short c2, short c3)
{
  int i;

  for(i=0; i<100; i++)
    y[i+1]=(c1*x[i] + c2*x[i+1] + c3*y[i]) >> 15 ;

}
```

# 4 Annexes

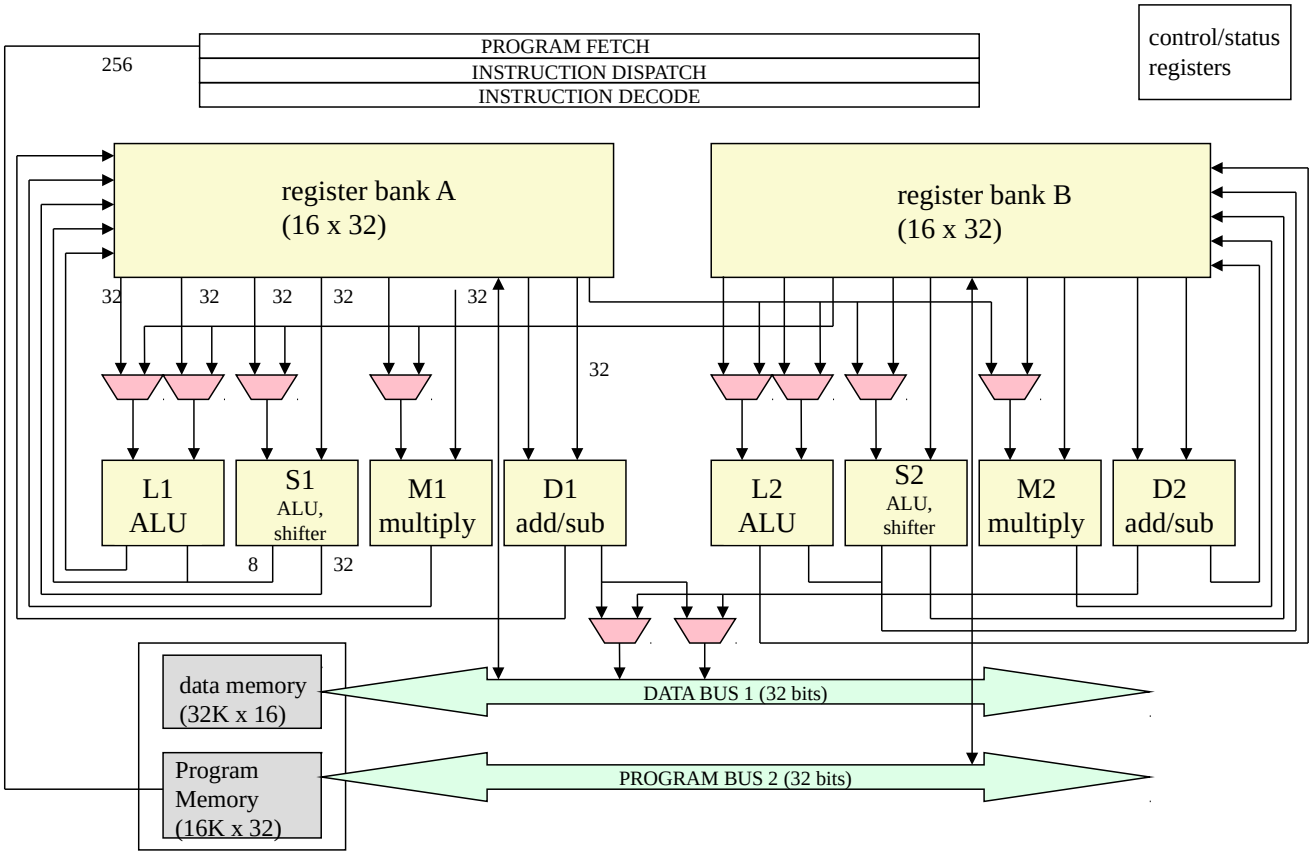


FIGURE 1 – Schéma fonctionnel du TMS320C62

| Type instruction | Latence |
|---|---|
| NOP (no operation) | 1 |
| Store | 1 |
| Instr. Simple | 1 |
| Multiplication (16x16) | 2 |
| Load | 5 |
| Branchement | 6 |
| Addition SP | 5 |
| Multiplication SP | 5 |

TABLE 2 – Latence des instructions

| .L Unit | .M Unit | .S Unit | | .D Unit | |
|---|---|---|---|---|---|
| ABS | MPY | ADD | SET | ADD | STB (15-bit offset)‡ |
| ADD | MPYU | ADDK | SHL | ADDAB | STH (15-bit offset)‡ |
| ADDU | MPYUS | ADD2 | SHR | ADDAH | STW (15-bit offset)‡ |
| AND | MPYSU | AND | SHRU | ADDAW | SUB |
| CMPEQ | MPYH | B disp | SSHL | LDB | SUBAB |
| CMPGT | MPYHU | B IRP† | SUB | LDBU | SUBAH |
| CMPGTU | MPYHUS | B NRP† | SUBU | LDH | SUBAW |
| CMPLT | MPYHSU | B reg | SUB2 | LDHU | ZERO |
| CMPLTU | MPYHL | CLR | XOR | LDW | |
| LMBD | MPYHLU | EXT | ZERO | LDB (15-bit offset)‡ | |
| MV | MPYHULS | EXTU | | LDBU (15-bit offset)‡ | |
| NEG | MPYHSLU | MV | | LDH (15-bit offset)‡ | |
| NORM | MPYLH | MVC† | | LDHU (15-bit offset)‡ | |
| NOT | MPYLHU | MVK | | LDW (15-bit offset)‡ | |
| OR | MPYLUHS | MVKH | | MV | |
| SADD | MPYLSHU | MVKLH | | STB | |
| SAT | SMPY | NEG | | STH | |
| SSUB | SMPYHL | NOT | | STW | |
| SUB | SMPYLH | OR | | | |
| SUBU | SMPYH | | | | |
| SUBC | | | | | |
| XOR | | | | | |
| ZERO | | | | | |

† S2 only
‡ D2 only

| .L Unit | .M Unit | .S Unit | .D Unit |
|---|---|---|---|
| ADDDP | MPYDP | ABSDP | ADDAD |
| ADDSP | MPYI | ABSSP | LDDW |
| DPINT | MPYID | CMPEQDP | |
| DPSP | MPYSP | CMPEQSP | |
| DPTRUNC | | CMPGTDP | |
| INTDP | | CMPGTSP | |
| INTDPU | | CMPLTDP | |
| INTSP | | CMPLTSP | |
| INTSPU | | RCPDP | |
| SPINT | | RCPSP | |
| SPTRUNC | | RSQRDP | |
| SUBDP | | RSQRSP | |
| SUBSP | | SPDP | |

TABLE 3 – Répartition des instructions entières et flottantes dans les unités fonctionnelles